POLYMORHISM

MODULE 7

Content

- Polymorphism
- Virtual Members
- Abstract Class

Polymorphism

> There are three major concepts in object-oriented programming:

1. Classes,
2. Inheritance,
3. Polymorphism, which is implemented in C++ by virtual functions.

Polymorphism

> In real life, there is often a collection of different objects that, given identical instructions (messages), should take different actions. Take teacher and principal, for example.

> Suppose the minister of education wants to send a directive to all personnel: “Print your personal information!” Different kinds of staff (teacher or principal) have to print different information.

> But the minister doesn’t need to send a different message to each group.

– One message works for everyone because everyone knows how to print his or her personal information.

Polymorphism

- Polymorphism means “taking many shapes”. The minister’s single instruction is polymorphic because it looks different to different kinds of personnel.
- Typically, polymorphism occurs in classes that are related by inheritance. In C++, polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- This sounds a little like function overloading, but polymorphism is a different, and much more powerful, mechanism. One difference between overloading and polymorphism has to do with which function to execute when the choice is made.

Polymorphism

- With function overloading, the choice is made by the compiler (compile-time).
- With polymorphism, it’s made while the program is running (run-time).

Member Functions Accessed with Pointers

```
class Square {
  protected:
    double edge;
  public:
    Square(double e):edge(e){  }
    double area(){return(edge*edge);}
};
class Cube : public Square {
  public:
    Cube(double e):Square(e){}
    double area(){
      return( 6.0 * edge * edge ) ;
    }
};
```

```
int main(){
     Square S(2.0);
     Cube C(2.0);
     Square *ptr;
     char c;
     cout << "Square or Cube";
     cin >> c;
     if (c=='s')
        ptr=&S;
     else
        ptr=&C;
     ptr->area();     // which Area ???
}
```

```
ptr = &C;
```

> Remember that it's perfectly all right to assign an address of one type (Derived) to a pointer of another (Base), because pointers to objects of a derived class are type compatible with pointers to objects of the base class.

> Now the question is, when you execute the statement

```
ptr->area();
```

> what function is called?

– Is it `Square::area()` or `Cube::area()`?

Virtual Member Functions Accessed with Pointers

> Let's make a single change in the program: Place the keyword **`virtual`** in front of the declaration of the `area()` function in the base class.

```
class Square {
  protected:
      double edge;
  public:
      Square(double e):edge(e){  }
      virtual double area(){return(edge*edge);}
};
class Cube : public Square {
  public:
      Cube(double e):Square(e){}
      double area(){
       return( 6.0 * edge * edge ) ; }
};
```

```
int main(){
     Square S(2.0);
     Cube C(2.0);
     Square *ptr;
     char c;
     cout << "Square or Cube";
     cin >> c;
     if (c=='s')
        ptr=&S;
     else
        ptr=&C;
     ptr->area();     // which Area ???
}
```

Late Binding

- Now, different functions are executed, depending on the contents of **ptr**.
- Functions are called based on the contents of the pointer **ptr**, not on the type of the pointer.
- This is polymorphism at work. I've made **area()** polymorphic by designating it **virtual**.
- How does the compiler know what function to compile?

Late Binding

- The compiler has no problem with the expression

```
ptr->area();
```

- Always compiles to the **`area()`** function in base class.
- The compiler doesn't know what class the contents of **`ptr`** may be a pointer to.
- It could be the address of an object of the **`Square`** class or the **`Cube`** class.
- Which version of **`area()`** does the compiler call?
  - At the time it's compiling the program, ***the compiler doesn't know what to do***
  - So it arranges for the decision to be ***deferred until the program is running***.

Late Binding

- At runtime, when the function call is executed, code that the compiler placed in the program finds out the type of the object whose address is in **`ptr`** and calls the appropriate **`area()`**:
  - **`Square::area()`**
  - **`Cube::area()`**
- Selecting a function at runtime is called ***late binding*** or ***dynamic binding***.
- By default, matching of function call with the correct function definition happens at compile time. This is called ***static binding*** or ***early binding*** or ***compile-time binding***.

Late Binding

- Late binding requires a ***small amount of overhead***
  - The call to the function might take something like 10 percent longer
  - But provides an enormous increase in power and flexibility.

How It Works

- A normal object with no virtual functions is stored in memory and contains only its own data
- When a member function is called for such an object
  - The compiler passes to the function the address of the object that invoked it.
  - This address is available to the function in the `this` pointer
  - The `this` pointer is used to access the object's data.

How It Works

- The address in `this`
  - Is **generated by the compiler** every time a member function is called
  - Is **not stored** in the object
  - **Does not** take up space in memory.
  - Is **the only connection** that's necessary between an object and its normal member functions.

How It Works

- With virtual functions, things are more complicated.
- When a derived class with virtual functions is specified, the compiler creates a table—an array—of ***function addresses*** called the **virtual table**.
- The `Square` and `Cube` classes each have their own virtual table.
- There is an entry in each virtual table for every virtual function in the class.
- Objects of classes with virtual functions contain a pointer to the virtual table of the class.
- These object are slightly larger than normal objects.

How It Works

- In the example, when ***a virtual function*** is called for an object of **Square** or **Cube**, the compiler, *instead of specifying what function will be called*, **creates code** that
  - First look at the object's virtual table
  - Then uses this to access the appropriate member function address.
- Thus, for ***virtual functions***, the object itself determines what function is called, rather than the compiler.

- **Example:** Assume that the classes **Teacher** and **Principal** contain two virtual functions.

```
class Teacher {
      string *name;
      int numOfStudents;
  public:
      virtual void read();
      virtual void print() const;
};

class Principal : public Teacher {
  string *SchoolName;
  public:
  void read();
  void print() const;
};
```

Virtual Table of **Teacher**

&Teacher::read
&Teacher::print

Virtual Table of **Principal**

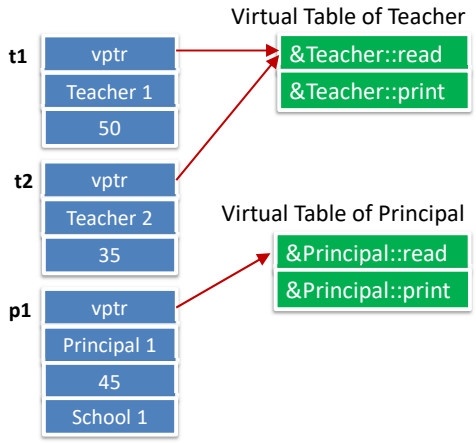
&Principal::read
&Principal::print

Virtual Table

> Objects of Teacher and Principal will contain a pointer to their virtual tables.

```
int main(){
 Teacher t1("Teacher 1", 50);
 Teacher t2("Teacher 2", 35);
 Principal p1("Principal 1",45,"School 1");
   :
}
```

Virtual Table



Virtual Table

> MC68000-like assembly counterpart of the statement

```
ptr->print();
```

> Here, **ptr** contains the address of an object.

```
move.l    ptr, this  ; this to object
movea.l   ptr, a0    ; a0 to object
movea.l   (a0), a1 ; a1<-vptr
jsr       4(a1)      ; jsr print
```

> If the print() function would not a virtual function:

```
move.l    ptr, this  ; this to object
jsr teacher_print
   or
jsr  principal_print
```

Don't Try This with Objects

> Be aware that the virtual function mechanism works only with pointers to objects and, with references, not with objects themselves.

```
int main(){
    Square S(4);
    Cube C(8);
    S.area();
    C.area();
}
```

> Calling virtual functions is a time-consuming process, because of indirect call via tables. Don't declare functions as virtual if it is not necessary

```
class Square {
  protected:
    double edge;
  public:
    Square(double e):edge(e){   }
    virtual double area(){
                  return(edge*edge);}
};
class Cube : public Square {
  public:
    Cube(double e):Square(e){}
    double area(){
      return( 6.0 * Square::area() ) ;
    }
};
```

Here, **Square::Area()** *is not virtual*

Homogeneous Linked Lists & Polymorphism

> Most frequent use of polymorphism is on collections such as linked list:

```
class Square {
  protected:
   double edge;
  public:
   Square(double e):edge(e){   }
   virtual double area(){ return(edge*edge); }
   Square *next ;
};
class Cube : public Square {
  public:
    Cube(double e):Square(e){}
    double area(){ return(6.0*edge*edge); }
};
```

```
int main(){
      Cube c1(50);
      Square s1(40);
      Cube c2(23);
      Square s2(78);
      Square *listPtr; // Pointer of the linked list
      listPtr=&c1;
      c1.next=&s1;
      s1.next=&c2;
      c2.next=&s2;
      s2.next=0L;
      while (listPtr) { // Printing elements of list
         cout << listPtr->area() << endl ;
         listPtr=listPtr->next;
      }
}
```

Abstract Classes

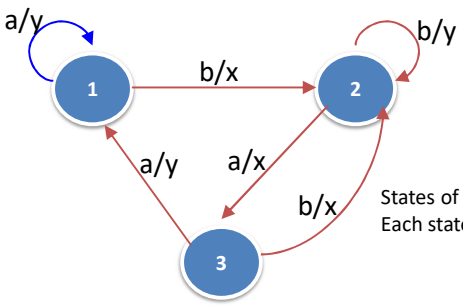
- To write polymorphic functions wee need to have derived classes.
- But sometimes we don't need to create any base class objects, but only derived class objects.
- The base class exists only as a starting point for deriving other classes.
- This kind of base classes are called an ***abstract class***, which means that ***no actual objects*** will be created from it.
- Abstract classes arise in many situations.
  - A factory can make a sports car or a truck or an ambulance, but it can't make a generic vehicle.
  - The factory must know the details about what kind of vehicle to make before it can actually make one.

Pure Virtual Classes

- It would be nice if, having decided to create an abstract base class, I could instruct the compiler to actively prevent any class user from ever making an object of that class.
- This would give me more freedom in designing the base class because I wouldn't need to plan for actual objects of the class, but only for data and functions that would be used by derived classes.
- There is a way to tell the compiler that a class is abstract: You define at least one ***pure virtual*** function in the class.
- A pure virtual function is a virtual function with no body.
- The body of the virtual function in the base class is removed, and the notation **=0** is added to the function declaration.

A Finite State Machine (FSM) Example

State : { 1 , 2 , 3 }
Input : { a, b }, x to exit
Output : { x , y }



States of the FSM are defined using a class structure.
Each state is derived from the same base class.

```
class State { // Base State (Abstract Class)
   protected:
     State * const next_a, * const next_b;
     char output;
   public:
     State(State& a,State&  b)
                    :next_a(&a),next_b(&b) { }
     virtual State* transition(char)=0;
};
```

```
class State1 : public State {
  public:
     State1(State&  a,State&  b) : State(a, b) { }
     State* transition(char);
};
class State2 : public State {
  public:
     State2(State&  a,State&  b): State(a, b) { }
     State* transition(char);
};
class State3 : public State {
  public:
      State3(State&  a,State&  b) : State(a, b) { }
      State* transition(char);
};
```

> The transition function of each state defines the behavior of the FSM. It takes the input value as argument, examines the input, produces an output value according to the input value and returns the address of the next state.

```
State* State1::transition(char input){
    switch(input){
      case 'a':  output = 'y';
         return next_a;
      case 'b': output = 'x';
         return next_b;
      default :
         cout << endl << "Undefined input";
         cout << endl << "Next State: Unchanged";
         return this;
   }
}
```

> The FSM in our example has three states.

```
class FSM {
  State1 s1;
  State2 s2;
  State3 s3;
  State *current;
 public:
  FSM():s1(s1,s2),s2(s3,s2),
        s3(s1,s2),current(&s1){
  }
  //Starting state is State1
  void run();
};
```

```
void FSM::run()  {
 char in;
 do {
    cout << "\nGive the input value "
         << "(a or b;  x: EXIT) ";
    cin >> in;
    if (in != 'x')
       current = current->transition(in);
       // Polymorphic function call
    else
      curent = 0; // EXIT
    }  while(current);
}
```

> The transition function of the current state is called.
> Return value of the function shows the next state of the FSM.

Virtual Constructors?

> Can constructors be virtual?

Virtual Constructors?

> Can constructors be virtual?

  No, they can't be.

> When you're creating an object, you usually already know what kind of object you're creating and can specify this to the compiler. Thus, there's not a need for virtual constructors.

> Also, an object's constructor sets up its virtual mechanism (the virtual table) in the first place. You don't see the code for this, of course, just as you don't see the code that allocates memory for an object.

> Virtual functions can't even exist until the constructor has finished its job, **so constructors can't be virtual**.

Virtual Destructor

> Recall that a derived class object typically contains data from both the base class and the derived class.

> To ensure that such data is properly disposed of, it may be essential that destructors for both base and derived classes are called.

Virtual Destructors

```
class Base {
   public:
     ~Base(){ cout << "Base destructor"; }
};
class Derived : public Base {
  public:
     ~Derived(){ cout << "Derived destructor";}
};
int main(){
   Base* pb = new Derived;
   delete pb;
   cout << endl << "Program terminates.\n";
}
```

Virtual Destructors

> But the output is

```
Base Destructor
Program terminates
```

> In this program **bp** is a pointer of **Base** type. So it can point to objects of **Base** type and **Derived** type. In the example, **bp** points to an object of **Derived** class, but while deleting the pointer only the **Base** class destructor is called.

Virtual Destructors

- This is the same problem you saw before with ordinary (non-destructor) functions.
- If a function isn't **`virtual`**, only the base class version of the function will be called when it's invoked using a base class pointer, even if the contents of the pointer is the address of a derived class object.
- **`Derived`** class destructor is never called.
- This could be a problem if this destructor did something important.

Virtual Destructors

```
class Base {
  public:
     virtual ~Base(){cout << "Base destructor";}
};
class Derived : public Base {
  public:
     ~Derived(){ cout << "Derived destructor";}
};
int main(){
   Base* pb = new Derived;
   delete pb;
   cout << endl << "Program terminates.\n";
}
```